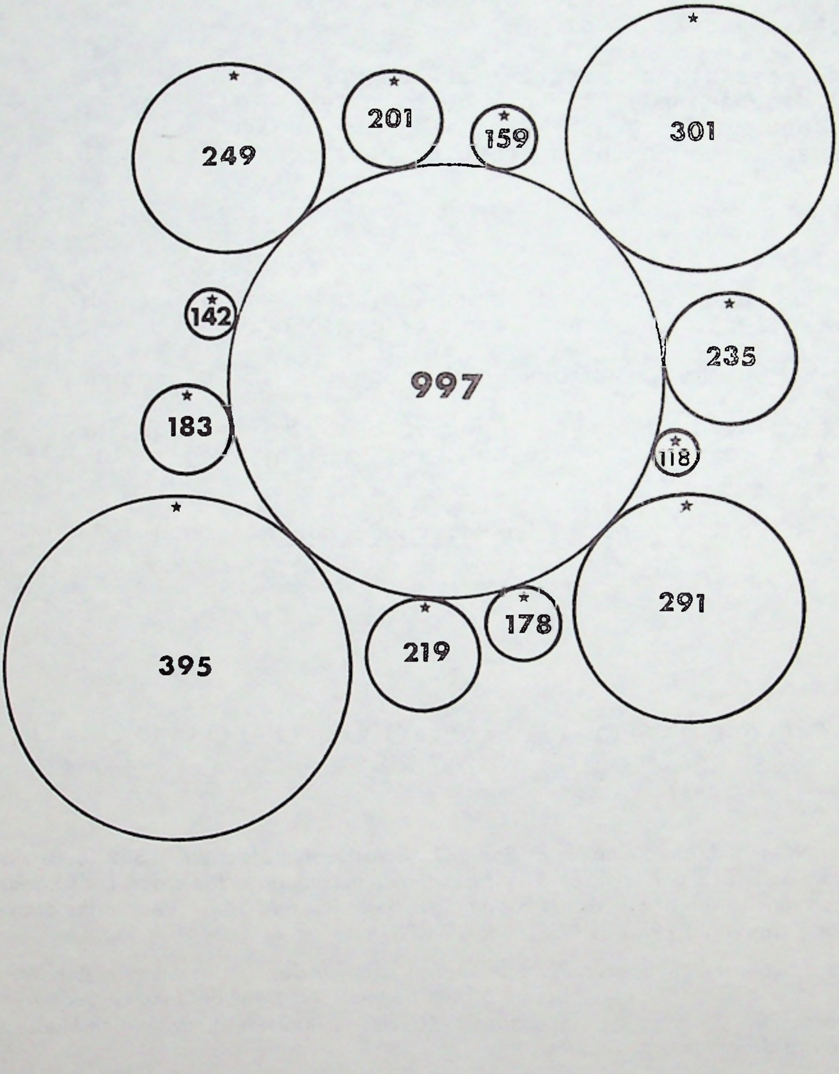
Gear Train



Popular Computing

JULY 1975 VOLUME 3 NUMBER 7

GEAR TRAIN

Twelve gears of various sizes are meshed with a large central gear as shown on the cover. The number of teeth on each of the gears is indicated. The train of gears rests with a point on each peripheral gear, indicated by a star, straight up.

The central gear starts rotating at 100 rpμ (revolutions per microsecond) at noon, coordinated universal time, January 1, 1975. When will the twelve peripheral gears again be in their starting positions simultaneously?

PROBLEM 94

For purposes of this problem, take the length of the year 1975, as of noon January 1, to be:

365 days, 5 hours, 48 minutes, 45.6170 seconds,

with each succeeding year shorter by .0053 seconds, and with each year assumed to have its length fixed as of January 1.

(Adapted from a problem in Modern Mechanix around 1934.) □

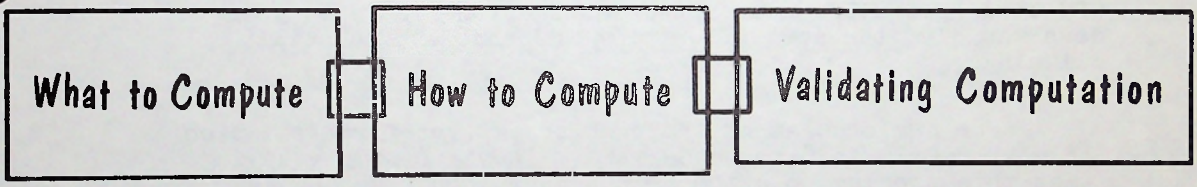
○ ○

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $18 per year, or $15 if remittance accompanies the order. For Canada and Mexico, add $4 per year to the above rates. For all other countries, add $6 per year to the above rates. Back issues $2 each. Copyright 1975 by POPULAR COMPUTING.

Publisher: Fred Gruenberger
Editor: Audrey Gruenberger
Associate Editors: David Babcock, Irwin Greenwald
Contributing editors: Richard Andree, Daniel D. McCracken, William C. McGee
Advertising manager: Ken W. Sims
Art Director: John G. Scott
Business Manager: Ben Moore

What Should Be Computed?

The art of computing can be divided into three broad areas:



The criteria for intelligent use of a computer

The mechanics of using a computer

Program testing

The areas overlap, not only in their concepts and principles, but also in time; that is, the computist must keep the precepts and practices of all three areas in mind constantly. In this series of articles, much attention has been paid to the third area--program testing. The second area is the subject of 90% of all the books published about computing. Something should be said about the first area---What to compute.

What constitutes a good computer problem? The other side of the coin is even more important; namely, what constitutes poor use of a computer?

The answer to "What?" can be reduced to a set of criteria. Let us illustrate with an old problem.

1

> A man cashes a check at a bank. The teller mistakenly interchanges the amount for dollars and the amount for cents, so that the man receives more than the amount of the check. The man finds that, after spending $14.19, he still has twice as much as the check called for. What was the amount on the check?

The first criterion is usefulness; that is, the problem solution must be useful. The tricky question is, useful to whom? If each person decided to work only on problems that he considered useful to him, personally, precious little would get done. Nearly everyone regards the other person's area of interest as almost totally useless.

In the real world, the criterion of usefulness is decided in a simple way: if you can afford it, you can have it. If what you want involves programming and CPU time, then you must be prepared to pay for them. The cost of CPU time (per million executed instructions, say) is going down rapidly; the cost of *writing* instructions is still climbing.

For our problem of the bank check, the first reaction of most people is "Who needs it?" They are simply saying that this problem solution is not useful to them and is therefore not useful at all, and cannot therefore constitute intelligent use of a computer if a computer solution is proposed.

In a classroom situation, the criterion of usefulness can be handled in other ways. If the proposed problem solution demonstrates some facet of computing (as we are doing right now) then it is surely useful, since that is what the class is there to learn. An instructor could go further, and satisfy the criterion of usefulness simply by declaring that the problem solution is useful (this technique saves time, but hardly constitutes good pedagogy). Or, it can be reasoned that the usefulness lies in learning how to handle another new situation; that every problem that has novel elements in it is useful to a student of computing.

But the game we're playing goes more like this: we have a problem to solve, and we must solve it. The question is, of all possible tools at our disposal, is the computer the proper one to use to achieve that end?

The second criterion is: the problem must be defined. This is essential whether or not a solution is to be attempted by computer. The difference with computer use (and the difference is small) is that it is necessary to define the inputs and outputs clearly, and the algorithm between them, in more precise terms than is customary with a solution by other means. Moreover, a computer solution to a problem should be defined completely before any part of it is coded. With computers, it is difficult and awkward to try to work out the solution as you go along. In any event, the definition of the problem must go far beyond vagaries such as "write a program to perform inventory control," or "can we program the computer to win at dice?" The situation is analogous to invoking "the law of averages" or of a student asking "Do you grade on a curve?"--both of which are ill-defined situations. If we don't know precisely what the problem is, then any subsequent scheme for solving it has slim chance of success. Or, as someone has said, if you don't know where you're going, then any road will take you there.

For our bank check problem, the definition is implied in an example:

```
Check      Man receives
58.87         87.58
              14.19  spent
              -----
              73.39 left
```

(and if the 73.39 were twice as large as 58.87, we'd have a required solution.)

The third criterion is: we must have $\boxed{\text{a}}$ method of solution. (Not necessarily the method, or even a good method).

Since the bank check problem sounds vaguely familiar, it is reasonable to inquire into the possibility of solving it by older, more traditional means (and if that would work, then we have taken care of the sixth criterion also). We might, then, try for an algebraic solution:

Let D equal the dollar amount of the original check.
C equal the cents amount of the original check.
The original check, in cents, is thus $100D + C$.
The man receives $100C + D$.

and we have then the equation:

$$2(100D + C) = 100C + D - 1419,$$

which follows quite directly from the statement of the problem.

Now we can see why the problem is only vaguely familiar. It isn't a problem in algebra at all. We have one equation in two variables, plus the extra condition that C and D must be integers; this is then a linear Diophantine equation to be solved and, while there are straightforward methods for doing it, they are not familiar to most high school graduates.

If an analytic solution is beyond us, and no algorithm leaps to mind, then we might try a standard crowbar approach; namely, to solve the problem by exhaustion. The <u>solution space</u> is the 10000 numbers from 00.00 to 99.99, and we could try every value in that range until we found one or more that satisfied the problem.

Here is where computer solutions begin to differ from the other tools we have used. The sheer speed and low cost-per-operation of the computer make exhaustive solutions feasible for the first time, even though such solutions may be inefficient and wasteful. The criterion at hand calls for <u>a</u> method of solution. We may try many methods for any given problem; it is not uncommon to find that a crude solution, implemented for a few cases on a computer, will lead us to a more sophisticated method of solution.

Before leaping to the coding sheets to implement the brute force solution just outlined, a few moments' thought will indicate some drastic shortcuts. The smallest possible value for the check, for example, is 00.15. The value of C must always be greater than the value for D (or there would be no problem). With these two notions in mind, there are no more than 4845 cases to run through. For each of them (rearranging our equation a bit) it remains to determine only whether or not

$$199D + 1419 - 98C = 0.$$

When the values D = 17 and C = 49 are reached, the answer is yes, and a solution has been achieved; the solution popped out on the 1474th case tried. If the program is written to halt at that point, it would not have been demonstrated that there is not another solution possible.

The problem is thus solved and, if done as described above, we have a deck of cards bearing the program that implemented our solution. The computer would have executed some 40,000 instructions during the solution, consuming about 1/25 second of CPU time, and costing about $0.0008 at current rates. The proposed solution thus satisfies <u>the fourth criterion: it must fit the available machine in space and time</u>. For the given solution, there would be 40 to 50 instructions in machine language, plus perhaps 10 data numbers (that's the space requirement) and the consumption, as noted, of 1/25 second of time. The proposed solution would fit any computer ever built.

The fifth criterion is repetition in the solution. This says simply that computers do best what they have already done; that we should capitalize on the machine's ability to repeat any set of instructions. In the extreme case in which a set of instructions is executed only once, the user would not know that the calculations are correct without duplicating their action by hand, in which case why use a computer? In practice, the amount of repetition may vary. The point is, the more, the better--that's what we get computers for.

The sixth criterion is cost-effectiveness; that is, would some other tool besides the computer provide the desired solution at less expense of time or money? There are many other tools available (desk calculators, punched card machines, graphical methods, analytic solutions, etc.) and one should not leap to the coding sheets for any and every information processing problem.

It is now time to introduce a second problem:

2

A man cashes a check at a bank. The teller mistakenly interchanges the amount for dollars and the amount for cents, so that the man receives more than the amount of the check. The man finds that, after spending $14.20, he still has twice as much as the check called for. What was the amount on the check?

The second problem sounds like the first, but there are two points of difference. The only parameter in the problem has changed by one cent and we have solved the first problem by an exhaustive search by computer. It would be tempting to change that parameter and re-submit the program.

Although digital computers are discrete devices, the principle of continuity applies to many situations. Given an arithmetic process of any sort wherein an input number X is transformed into an output number Y, then a small change in X should produce a small change (not necessarily the same change) in Y (except if the change in X induces a division by zero). Thus, the fact that the input number in the first problem has changed only .07% makes it reasonable to conclude that the output number should be in the neighborhood of D = 17, C = 49. Values like the following should be tried:

17.50
17.48
16.49
18.49

and a few minutes of hand calculation should reveal the solution. However, the second problem is not a computer problem for a different reason: it is not really a proper problem, since it has no solution. That is, there are no integer values for C and D that will satisfy the conditions of the problem. If we had tried to solve it by re-using the program, and we had not remembered to terminate that program gracefully (when D is 98 and C is 99) the program would probably have hung up in an endless loop.

This leads us to a third problem:

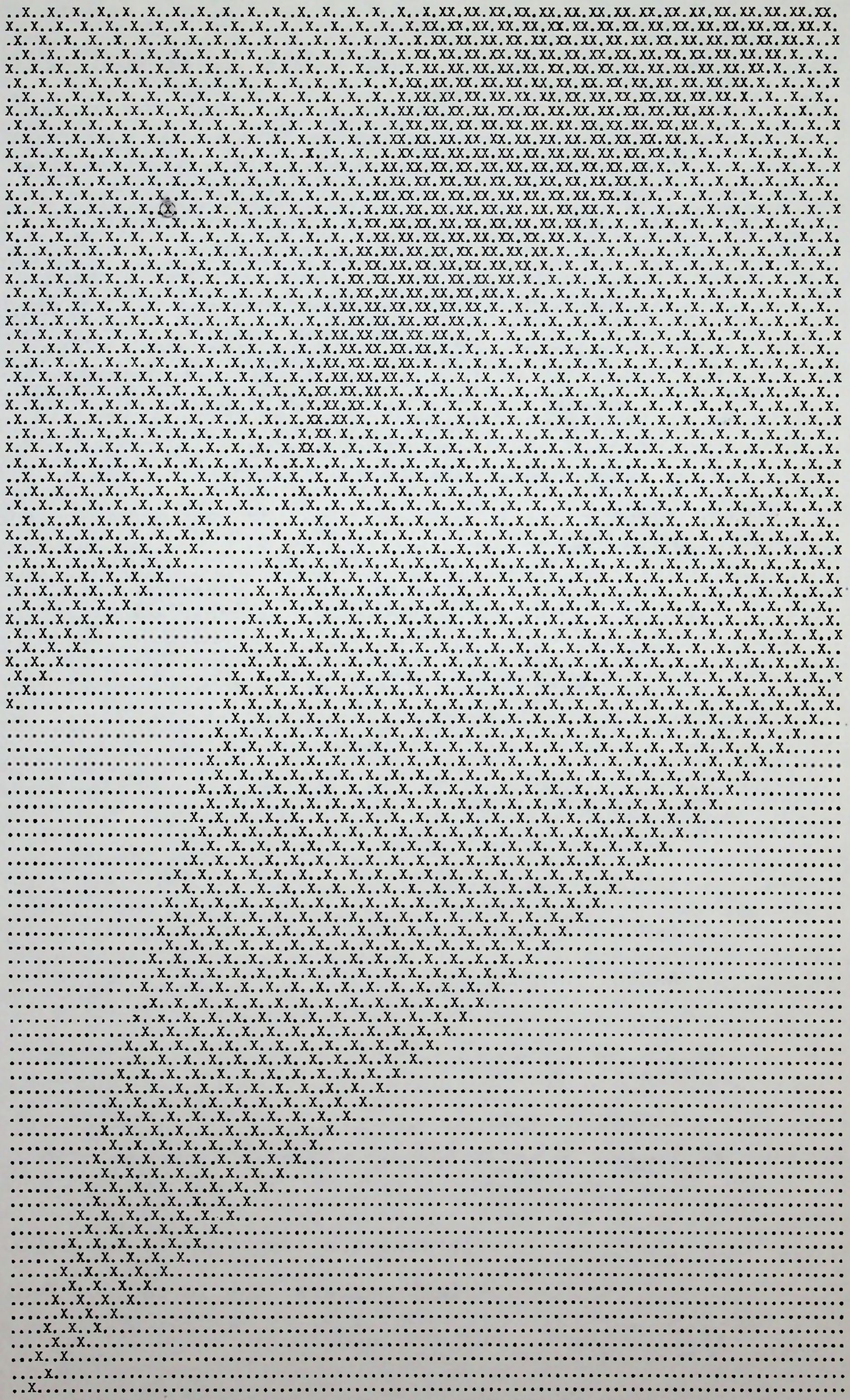
3

> A man cashes a check at a bank. The teller mistakenly interchanges the amount for dollars and the amount for cents, so that the man receives more than the amount of the check. The man finds that, after spending X dollars, he still has twice as much as the check called for. What are the values of X that make this a real problem?

The accompanying chart shows the complete solution to the third problem. There are 100 positions vertically, representing dollar amounts for X from 00 to 99, and 100 lines horizontally, representing cents amounts for X from 00 to 99. Where the letter X appears, the value is valid and a solution is possible; the appearance of a period denotes an invalid value and no solution is possible. The solution for the value 14.19 for the parameter is circled. (In a subsequent article, analytic methods for solving the problem will be discussed.) □

* *

Log 28	1.44715803134221922113969404804162224701995215924818
Ln 28	3.33220451017520392393981698635953286578808499830237
$\sqrt{28}$	5.29150262212918118100323150727852085142051836616490
$\sqrt[3]{28}$	3.03658897187566251942080957850566963558145397724811
$\sqrt[5]{28}$	1.94729436123033627305241021184266846683871762199454
$\sqrt[10]{28}$	1.39545489401497183364071746773353720008309345383177
$\sqrt[100]{28}$	1.03388344273197337272579684565744678855189601119437
e^{28}	1446257064291.47517367704742299692885690206232950992 28751654790521104885742402301048039401
π^{28}	83214007069229.6122606377257364258820167204785440609 0072245815197480428001626905150489694
$\tan^{-1}$ 28	1.53509721411557264984463831856508032034163505803975



The Dollar Bill Problem

The serial numbers on dollar bills are 8 digits long, which means that there are 100,000,000 possible numbers. Of these, there are the following types:

Type		Number
ABCDEFGH	(all digits different)	1,814,400
AABCDEFG	(exactly 2 digits alike)	16,934,400
AAABCDEF	(exactly 3 digits alike)	8,467,200
AAAABCDE	(exactly 4 digits alike)	2,116,800
AAAAABCD	(exactly 5 digits alike)	282,240
AAAAAABC	(exactly 6 digits alike)	20,160
AAAAAAAB	(exactly 7 digits alike)	720
AAAAAAAA	(all digits alike)	10

These types total 29,635,930, leaving many types unaccounted for, including:

AAAAAABB	AAAAABBB	AAAAABBC
AAAABBBB	AAAABBBC	AAAABBCD
AAAABBCC	AAABBBCC	AAABBBCD
AAABBCCD	AABBCCDD	AABBCCDE
AABBCCCD	AABBCDEF	AABBBCDE

Problem: account for the remaining 70,364,070 cases.

(The corresponding breakdown of the 100,000 5-digit numbers is this:

ABCDE	30240
AABCD	50400
AAABC	7200
AAAAB	450
AAAAA	10
AAABB	900
AABBC	10800)

Determine the breakdown of the 1,000,000 6-digit numbers, the 10,000,000 7-digit numbers, and the 10^K K-digit numbers.

symposium 16

The 16th session of the annual invitational one-day computing symposium was held in San Diego November 10, 1974. The attendees were:

PROF. RICHARD ANDREE, University of Oklahoma
PAUL ARMER, Center for Advanced Studies in the Behavioral Sciences
CHARLES L. BAKER, Ocean Data Systems, Inc.
MORT BERNSTEIN, System Development Corporation
HOWARD BROMBERG, International Computer Trading Corporation
RICHARD CANNING, Canning Publications, Inc.
BRUCE GILCHRIST, Columbia University
GEORGE GLASER, AFIPS
IRWIN GREENWALD, Xerox Data Systems
PROF. FRED GRUENBERGER, California State University, Northridge
DR. RICHARD HAMMING, Bell Laboratories
JERRY KOORY, On-Line Business Systems
DAN McCRACKEN, Author
JULES MERSEL, City of Los Angeles
EDWARD STRAUB, Rockwell International
ROBERT SWANSON, California State University, San Diego
ROBERT WHITE, Informatics Inc.

The topic was The Future of Programmers, and each attendee had been furnished a copy of the essay on that subject that appeared in Issue 19. Each attendee also received a copy of a summary of the SILT report on the same topic, prepared by Mort Bernstein.

Following are excerpts from the transcript. The complete transcript can be obtained for $10 from the Bureau of Business Services and Research, California State University, Northridge 91324. Please note that the remarks that follow are taken out of context.

BROMBERG: At least as far as application programmers are concerned, the future is dim. It is analogous to someone wanting to become, today, a telephone operator. We have no need for them, and we'll replace them. I'm talking about programmers who serve the computer users. Their future is dim because of the expense. They are in a labor-intensive activity in an area where the cost of executed logic is falling rapidly, and the result is an imbalance. The cost of putting a simple payroll on the machine is now several orders of magnitude greater than the cost of the machine itself. One of these days—soon—management will no longer put up with that. We will all have to look for other ways to solve the problem of creating user applications. It cannot be forever done the way we are accustomed to doing it: namely, with people.

HAMMING: I suggest that we are proceeding down the same path we travelled with automobiles and other complex devices that have been placed in the hands of the masses; namely, that we remove more and more of the dials and levers until it is very easy to use.

BAKER: The statement was made in 1900 that by 1960 every man, woman, and child in the country would have to be a telephone operator. As it turned out, by 1960 every man, woman, and child in the country *was* a telephone operator. The definition of what was meant by "operator" changed very slowly; the user's needs were simple, and they inserted a lot of mechanics to make it easy for him to be an operator.

McCRACKEN: One of the reasons why Fred is so badly mixed up on this subject is that he assumes that programming is always going to be what it has been.

I brought a computer along with me. You've heard the statement that 90% of all the scientists who have ever lived are alive today. Here's a similar statement: 90% of all the processors ever made were shipped by Intel in the last two years—and they look like this tiny chip. Intel has thousands of customers, and all of them have to do what Bromberg referred to as hard wiring; they're programming read-only memories. It is very demanding programming, because it must be correct. If you ship 10,000 point-of-sale recorders, and one bit is wrong in the program, it costs a lot to replace all those chips. They tell me that only a small fraction of that work is being done by people who have been trained as programmers. They certainly aren't called programmers by the organizations that employ them. It's an example of how the type of job that a programmer does will almost certainly change. They're called logic designers or applications people, or whatever.

What Fortran did was open up computer applications to a lot of people who couldn't program at all before, but for the person who was already programming, Fortran did not increase his productivity by very much.

GLASER: You're talking then about a very elite group of people who do not get much benefit from the use of a higher level language.

BAKER: The people whose productivity did not increase were those who already knew how to do structured programming in absolute binary, which includes about five people in this room. The higher level languages contain a lot fewer techniques to implement these principles: COBOL, for example, has almost none.

GREENWALD: The reason you don't get the productivity increase out of the higher level languages is that programmers don't spend a lot of their time actually writing programs. Studies have shown that if you could reduce coding and checkout time to zero, you would get around a 2.98 increase in productivity; that is, the total amount of checked out programs produced in a given time would be less than three times as much. He spends a lot of time talking to customers, and around half his time maintaining programs.

KOORY: There are going to be two classes of programmers. One of them is the class we now call systems programmers; they're the tool and die makers of our trade; they make the compilers and the Mark IV systems and the tools that let the problem solvers solve their problems; they're logic designers. Their future is closely related to the hardware. Their career path is either technical or managerial, in managing other systems programmers. They may work for a vendor, or a software house, or a specialty house. I have no idea how many of them we'll need, but I think there will be more of them ten years from now than we have today. If we have 20,000 of them today (just as an example), ten years from now there will be 50,000.

The other class of programmers (which we may not call programmers in the future) are the problem solvers. They translate some problem that they are given into systems and methods that are applied by the computer. Their career path concerns the business that they're in, not computing as such. Their education should deal primarily with their business, rather than with programming or computing. Their path to president of the company parallels their company's business, rather than computing itself.

McCRACKEN: Let me insist that there is an activity in programming that is not coding, or problem solving, or applications work, or systems analysis. I prefer the term "program design," which is typified by such activities as drawing flowcharts (if you still use flowcharts). It is the stage between the work of the applications systems analyst, who specifies what it is that is to be done, and the work of coding. Program design has specialized techniques that we are currently learning. It consumes 3/4 of the programmer's time, together with a little time spent coding, and very little time spent on checkout, if the design stage is done properly. The hard part is the program design; coding is not a trivial task, but it becomes the small task. The person who does all this is a programmer, whatever his job title may be. The title will not, however, be "accountant," or "engineer." The job involves a set of techniques which can be learned and which are not simple. It involves a statement of the logic and the sequence of operations.

GRUENBERGER: All of which is great, when you start with the natural clear thinking of the McCrackens and the Bakers and the Bernsteins. But the next step—transfer of those techniques to run of the mill clods—is a big one, and I question the magic of the shark's teeth you're wearing.

GREENWALD: We have all assumed that programming is a profession. I think that it's many professions. For example, we now have experts in a strange language called JCL, and they sit in shops and help people get their jobs on the machine. They get smarter with time, and eventually build JCL procedures that are parameterized. We have languages like SIMULA and the extensible language at Harvard, which enable some people to create languages for other people to do their work. Then there are the people who build the SIMULA's; the people who are experts at data management; the people who spend their time on big operating systems, and those who worry about distributed operating systems. And there are those who specialize in real-time-applications.

It seems to be that there are many disciplines. I can't keep up with all of them, other than to know they exist. It's not clear to me that there can be a set of courses to train all of them. What I see coming out of the schools, with bachelor's degrees in computer science, is people who can't read, can't write, and can't think. They believe that the ultimate test is a program that works. They are un-read; they don't know who is notable in the field; they don't know what to read. One thing I think they should be taught is where the real costs of computing are. The real costs aren't in development; they're in post-development, especially for a vendor, but also in big business.

BERNSTEIN: To get to where we are now has taken the efforts of several hundred thousand people (let's not worry about whether they're called programmers). Now, as new hardware systems emerge, we'll need at least as many people to do our old work all over again, plus more people to deal with new things that are designed to fit the new systems. There are actually three areas: doing the old applications over; keeping the old systems operating; and working on the new systems. I see nothing to convince me that there will be a significant increase in productivity, of those of us here, or of anyone else now in the field, or of anyone coming into the field in the next five years or so. Therefore, I conclude that, just in terms of the sheer bulk of the programs that will have to be written, we'll need a lot more people. Perhaps we won't call them programmers. I don't see the automaticity coming along, nor any increase in productivity.

McCRACKEN: I think that one of the reasons why Fred comes to the wrong conclusion is that he thinks that applications will always be what they are now.

GRUENBERGER: You and Hamming are always putting words in my mouth. In this case, though, you're right. Processing a file of records for an apple warehouse is no different, as far as programming is concerned, from processing a file of records in a doctor's office. A file is a file is a file. If you can show some interesting new twist to some new application, we'll program *that* and it will apply to another thousand applications.

McCRACKEN: But there may be 10,000 applications to which it doesn't apply.

GRUENBERGER: But what is new and different in those applications? I'll say it again: as soon as you come up with something that is different, we can program that, too. All these nice new applications sound great, but the data processing that is involved in them is the same old stuff we've done for nearly 20 years now.

BAKER: I'm in a small scientific house, and one of our problems, besides that of growing bigger, is trying to get new work. We've come to the realization that such a company depends on the average programmer. If you base it on master programmers, then when they leave, you're dead.

MERSEL: The group assembled here may be the wrong one to consider the question raised at this symposium. The young people now entering our field have a stronger reason to look ahead than we had. The ones I speak to are studying different fields and *also* studying programming; they don't think too highly of programming by itself. They're using the computer as casually as we used a slide rule, and they're producing as much code as we used to.

BAKER: The feedback from employers, students, and the schools all operates toward the institutionalization of the craft, or trade we are in. It makes people fungible, which is a way of saying that you know what a guy can do because he has a label on him. His training may have been good or bad, but it's standardized, and a business can rely on a continuous supply of the same product. In our industry, as in any other, Gresham's law applies; the things you can rely on tend to be at the lowest common denominator. The bad programmers drive out the good because there are more of them; they're cheaper; they're more reliable—that is, you can rely on their (mediocre) performance, and you can budget for it and plan on it. You can justify all the costs associated with such people.

McCRACKEN: It's like a Howard Johnson's; whatever the quality level; you can depend on its uniformity.

BAKER: I have one more comment relating to the future of programmers: "There is a strong atavistic impulse in humans to do things for recreation that mankind formerly had to do for survival." If we have had to do programming for survival, perhaps we can look forward to a new renaissance in the recreational areas. □

The discussion on The Future of Programmers began with the essay that appeared in issue No. 19. The essay formed the basis for debate at the 16th one-day symposium (reported elsewhere in this issue). The following is by R. W. Hamming of the Bell Laboratories, who was one of the attendees.

The Future of Programmers

What is programming and who are programmers? Computer experts pride themselves on their clear thinking and expect it in others, yet they are not clear about how to answer these questions. Let us talk around the questions to see if we can find some illumination.

First, "What is programming?" When a person uses a telephone and presses the various buttons to call a number, he sends parameter values to an electronic telephone central office, which are inserted into already written, rather complex, programs. We usually do not feel that this is programming. When an airline reservations clerk uses a terminal all day to place a wide variety of reservations, we again usually do not feel that this is programming a computer, but is merely selecting particular programs and inserting certain parameters into them. But at the other end of the programming scale, things which we believe are programming appear to the operating system and assembler (assuming it is in "machine language") as merely inserting parameters. Thus there appears to be no easy, logical separation between programming and nonprogramming--at best it is a matter of degree.

But perhaps it is not *what* is done, but the *way* it is done that makes programming. Following this path, many computer experts boldly assert that programming is creating, selecting, and using algorithms; they also assert the converse, that the creation of algorithms makes it programming. In practice, this latter view would claim almost all of science; it is like saying that the use of mathematics is part of mathematics, and thereby claiming very large parts of most sciences as subparts of mathematics. This view may flatter the mathematician, but it hardly agrees with conventional usage, and hence is apparently not very useful.

Returning, therefore, to our original problem, if we cannot define the act of programming, perhaps we can define a programmer. As with the airline reservations clerk, the mere use of a computer clearly does not make the person more than a clerk. Again, there appears to be no widely satisfactory definition. But there is an interesting approach to the question just the same; even if we cannot define it, perhaps we can give a lower bound below which we do not consider a person to be a programmer.

If we wish to define "programmer" to have a moderate degree of dignity and stature, to approach professionalism if not reach it, then let us look at the three classic characteristics of a professional discipline: it requires extensive special training, experience is necessary and improves performance, and finally there is a code of ethics. With the rapid spread of better designed systems and languages, the strong current trends toward machine independence such as virtual memories, and with better documentation I suggest that a few courses in programming will provide most of what the average person will need to use computers in a particular field of application. Of course he may also need six courses in numerical analysis, three courses in accounts receivable, four courses in bridge design, etc., but it is not the expertise in the field of application we are talking about, it is the training in the use of the computer itself.

Thus we come to what many people consider the essence of programming and being a programmer--the design and construction of large software systems. Unfortunately for those who hope to spend their lives doing this, we have recently produced (after years of promises and a few results) compiler-compilers (and other software tools to build software) which are practical and are in daily use.

This late development of the use of the computer by the computer experts themselves is the same phenomenon as is seen in other fields. There is the well-known saying, "The shoemaker's children go without shoes." Doctors often do not follow the advice they give to patients, lawyers leave estates in a messier state than the average person whom they advise to plan carefully, IBM was slow in using accounting machines for their own work. Similarly most programmers, while willing to use peripheral aids to debugging, stubbornly refuse to let the machine find and correct the bugs. In the past programmers have fancied themselves individual "artists," working in a cottage craftsman style, to whom it was an insult to suggest that a machine could do a better job more cheaply.

Well, the bell has finally tolled. In spite of most programmers we are moving into a world in which almost all of the software will be machine built. Faced with a new line of computers, designed more carefully than in the past to be compatible; not controlled by salesmen, machine designers, or by software experts, but by responsible systems engineers and executives who can see the whole problem from manufacture, to selling, to use, to the very expensive problem of years of field maintenance of software and hardware; we will resort to an elite team of programmers to create the millions of lines of code required. They will use the machine as their principal tool. They will survey the *whole* of the software to be built, decide on a general style for all of it, consider the rules they want to impose and the rules that are apparently imposed from the outside, and from their long experience with the specific languages and other processors, they will be able to make shrewd guesses at the costs of various minor features, and finally come to rational conclusions about when to violate the previous standards, and when to adhere to them. No doubt they will use simulations on the machine itself to answer some of the questions that arise. Finally, they will select with great care the software tools, their organization and interrelationship, and then let the machine generate the millions of lines of software required, along with a reasonable quantity and quality of documentation. The details of the software will not pass through any human mind in the process. We can automate the building of software in analogy with the hardware where we are approaching the point where most of the details of the printed circuit chips are not examined closely by engineers but are computer generated, laid out, drawn, photographed, reduced, and converted to masks for manufacture.

Of course there will be further stages, including carefully controlled tests of the software. When faults are found, the software will be redesigned and regenerated until a satisfactory final result is found--not perfect to be sure, but probably well below one bug per million lines of field issued code.

We observe in mature fields of research that a few experts tend to dominate the production of ideas, papers, and citations to their works. This phenomenon is less true in development work, but is still present. What can we expect in the research and development of software where the computer itself can fill the role of the assistants? Clearly, more than perhaps in any other development field, a few experts can, and will, dominate the whole picture. The "master programmer" will be the hero.

How many such highly trained, long experienced, very intelligent programmers will the whole country need, allowing for the myriads of various new applications and special languages? Remember we already have compiler-compilers which can be put in the hands of the expert in the field of application and whose use does not require an expert in programming. Remember, too, the degree of machine independence we are gradually achieving. From these facts it is difficult to believe that 10,000 experts will be needed. Allowing for a misuse of manpower by a factor of 10, we find that well below 100,000 programmers will be needed. Regretfully, we now have many more on hand who fancy themselves programmers. Fortunately, many of them are in fact partial experts in some field of application, and if they will pursue training in that field they can still be employed. But if they persist in seeking out only those jobs which are centered around the usual software then most of them will inevitably find themselves unemployed.

This is not merely an opinion of mine, it is an opinion of many others who have been programmers and now see the larger picture. The people who hire and fire programmers are determined that the past software follies do not happen again. Apparently the only way to achieve the future we want is to make the machine create the required software. To do this we will use the guidance of a small, elite, experienced crew who have deep training in software techniques. Of necessity they must be well trained in mathematics and capable of the high degree of abstraction necessary to see at one time all of the software as well as its development, its use, and its maintenance.

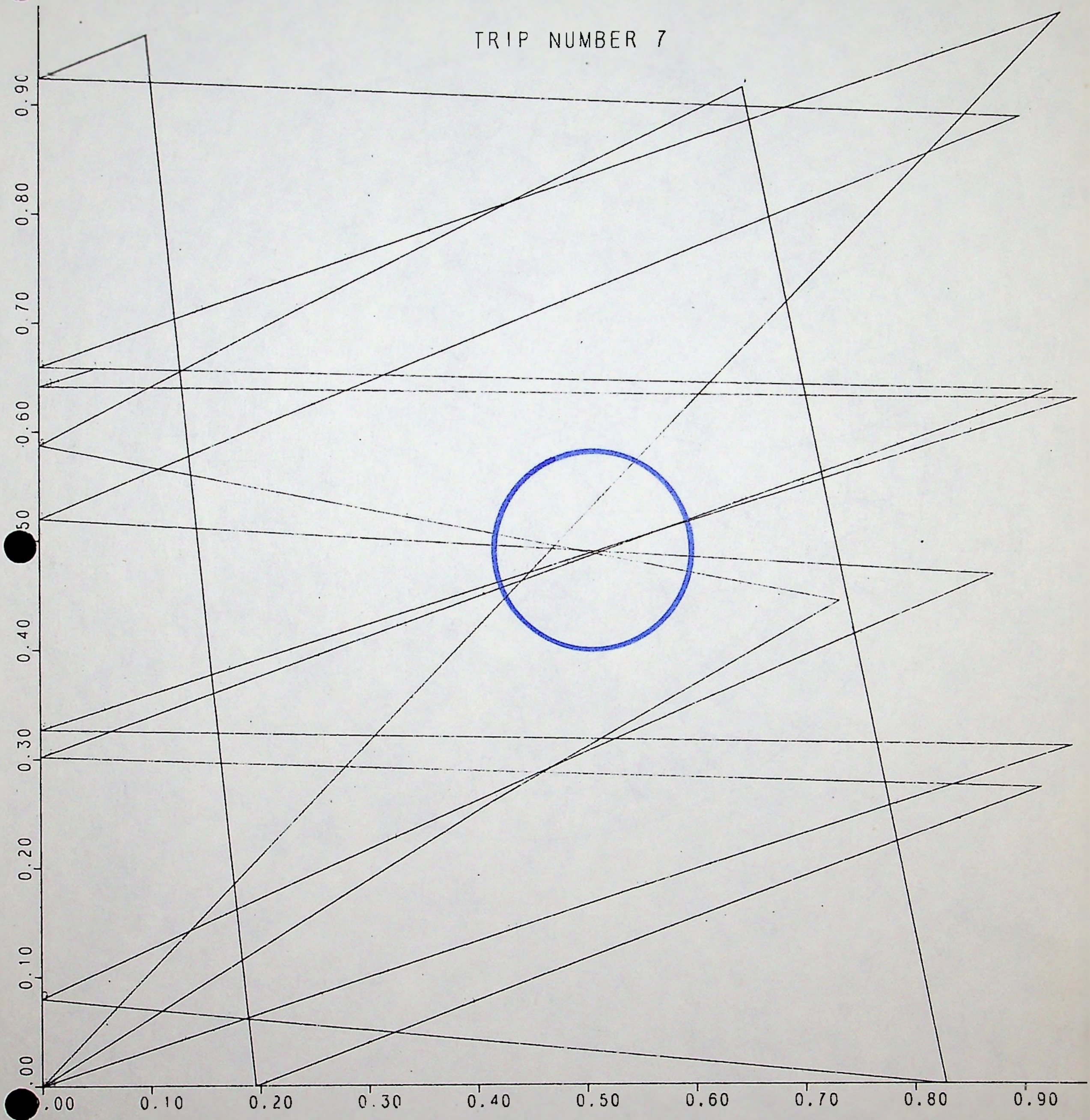
The message to the average student entering programming today is simple: either become an expert in some field of application or else be prepared to go very, very deeply into computer science. I don't know in detail what you will need to know; you are entering a rapidly evolving field, so concentrate on fundamentals and try to avoid small technical details of passing importance. This seems to suggest that you include lots of abstract mathematics so you will be able to keep up with the abstract theories of software generation that will evolve over your lifetime. But also include some of the humanities because it is the man-machine relationship that we care about. If you do not do this, you are probably starting a dead-end career. □

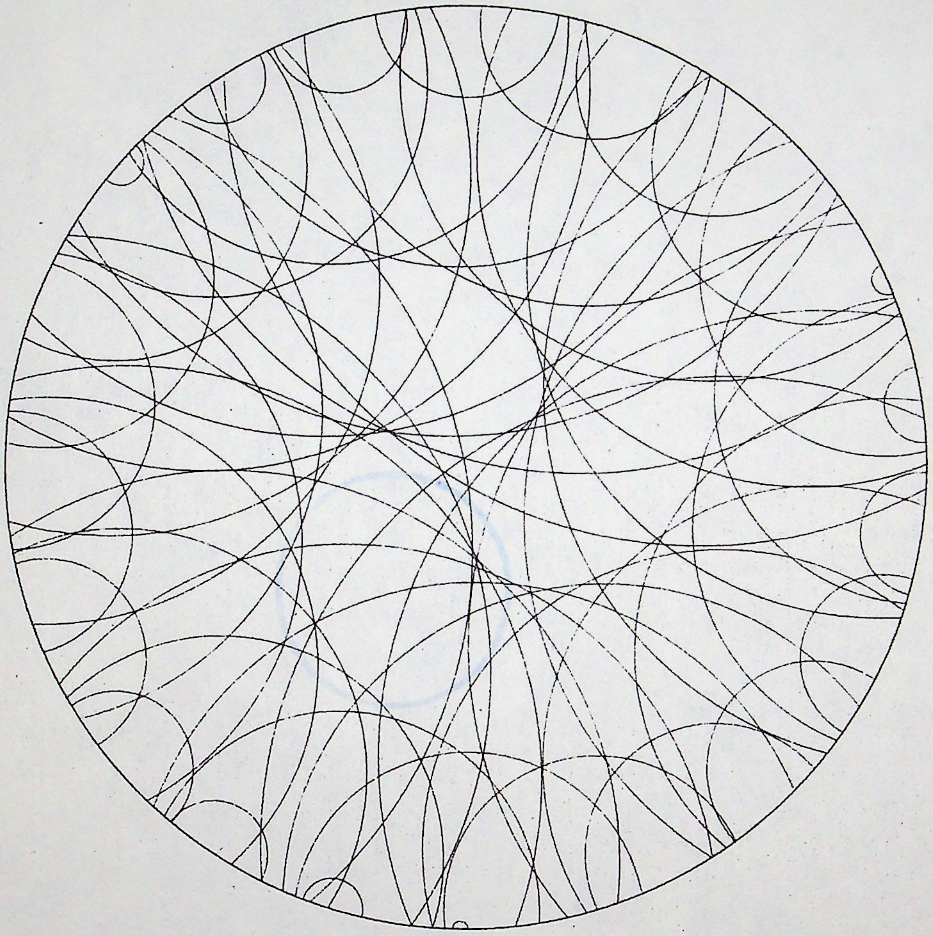
The plot on page 17 represents a complete solution to Problem 58, Another Route, from issue 18. The square root and cube root of successive integers, starting with 1, are calculated, and the mantissas are taken as coordinates in the unit square. The points so generated are joined, and the Problem was to determine how many times these lines cross during the first 100 legs of the trip. The result, found from the plot and from detailed analysis, is 68. Both parts of the task were done by Dorothy Cady, who pointed out that the data given for the problem (PC18-2) contained two last-digit errors. With mantissas rounded to 5 places, the square root of 11 should be .31662 and the cube root of 13 should be .35133.

The plot was made on the Calcomp 936 plotter at California State University, Northridge. The Fortran program for the plot was executed on the CDC 3170 at Northridge, producing a data tape for the HP-2100A mini computer which in turn drives the plotter. Much of the software for plotter control was written by David Babcock. The 936 plotter handles 1800 plot commands per second with .002 inch resolution. A variety of paper, pens, and colors of ink are available to the users.

While the 68 crossing points may be reasonably clear from the plot shown for the given problem, the situation might be highly ambiguous in general. For example, the area circled on the plot contains 10 crossings, but a slight shifting of the lines would make the fact difficult to ascertain.

The plot on page 18, also by Dorothy Cady, is a complete solution to Problem 87, The Obfuscating Circles, from issue 25. The large circle has a diameter of 6 units. The smaller circles have diameters of .1, .2, .3, .4,... units and are centered at points on the large circle that are 6 units apart. By plotting the successive circles, Mrs. Cady showed that 62 circles are needed to completely cover the large circle.


TRIP NUMBER 7
0.00
0.10
0.20
0.30
0.40
0.50
0.60
0.70
0.80
0.90



--Dorothy Cady 1975

Popular Computing